ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

setFTs: A package for Fourier transforms on set functions and its application to compiler flag optimization

Bachelors Thesis

Simon Ebner

28.08.22

Chris Wendler
Department of Computer Science, ETH Zürich

Abstract

Set functions are defined as functions over the domain of subsets of a finite set and map in most cases to the set of real numbers R. Set functions occur naturally in many applications when we're trying to assign a value to a subset. For example when measuring the information gathered restricted to only a subset of a set of sensors. In recent years there have been advancements in extending the concept of Fourier transform to set functions. This paved the way for the developement of multiple new algorithms. In this thesis we present "setFTs" a Python library that composes these algorithms into one applicable package. We will then showcase and demonstrate its features on a case study on compiler flag optimizations.

Abstract

Mengenfunktionen (set functions) sind definiert als Funktionen, über den Bereich der Teilmengen einer endlichen Menge. Sie kommen überall da vor, wo wir einer Teilmenge einen bestimmten Wert aus der Menge der reellen Zahlen R zuordnen wollen. Zum Beispiel beim Messen von der Information die von einer Teilmenge von Sensoren gesammelt wird. In den letzten Jahren, kam es zu neuen Erkentnissen bei der Erweiterung des Konzepts der Fourier-transformation für Mengenfunktionen. Dies hat den Weg bereitet für einige neue Algorithmen die auf diesem Konzept basieren. In dieser Arbeit präsentieren wir die Python library "setFTs", welche diese Algorithmen in ein einfach anwendbares Paket zusammenfasst. Wir wenden dieses dann auf das Beispiel der Compiler Flaggen Optimisation an, um zu demonstrieren wie es funktioniert.

Contents

Contents **3**
0.1 Introduction . . . 4
0.1.1 Motivation . . . 4
0.1.2 Contributions . . . 5
0.2 Signal processing with set functions . . . 6
0.2.1 Set functions . . . 6
0.2.2 Shifts . . . 6
0.2.3 Filters . . . 7
0.2.4 Fourier Transformation . . . 8
0.2.5 Convolution Theorem . . . 8
0.3 Algorithms and Functionalities . . . 9
0.3.1 Fast Fourier Transforms . . . 9
0.3.2 Sparse Fourier Transform . . . 11
0.3.3 Minimization . . . 16
0.3.4 Shapley Values . . . 18
0.4 Implementation . . . 20
0.4.1 Package Structure . . . 20
0.4.2 setfunctions module . . . 21
0.4.3 Plotting utilities . . . 22
0.4.4 Unittests . . . 26
0.4.5 Installation . . . 26
0.5 Experiment on Compiler Optimizations . . . 27
0.5.1 Experiment setup . . . 27
0.5.2 Results . . . 27
0.5.3 Shapley values . . . 28
0.6 Future Work . . . 29
0.6.1 Extending the Library . . . 29
0.6.2 Research on Compiler Optimizations . . . 29
0.7 Addendum . . . 30

Bibliography **33**

0.1 Introduction

This chapter serves as an overview over the topic of this thesis and highlights the importance of it.

0.1.1 Motivation

Set functions are an important family of functions for applications that are assigning a value to subsets of a finite set. For example in [9], where the problem statement of finding which subset of sensors produces the highest information gain, can be formulated as a maximization problem on a set function. Markus Püschel and Chris Wendler introduced in [12] the theoretical foundation for signal processing on set functions, and with it a definition for Fourier transform on set functions. They consider 5 bases defined over different shifts, that each result in a different notion of the Fourier transform. On that theoretical basis [16] introduces a different algorithm for the Fourier transform that focuses on Fourier sparse set functions. We also include the extension of that algorithm to the different bases from [6] and a maximization method from [14]. We then showcase our library on the example of compiler flag optimization.
A compiler's main purpose is translating high-level programming languages such as java, C++, etc. into lower level programming languages which a processor can then execute. It's secondary purpose is optimizing the written code into code that can execute more efficiently. Every modern compiler provides a multitude of optimizations. These optimizations are able to provide significant speedups to most programs, by changing a program into a semantically equivalent program that executes faster but provides the same results. The cost of these optimizations is generally negligible, as a small increase in compile time, can lead to greatly improved performance at runtime. A common example for such an optimization is loop-invariant code motion, where code that would be executed in each loop-iteration, but will always evaluate to the same value because it is independent of the loop variable, will be moved outside of the loop and only calculated once. All modern compilers utilize a combination of optimizations. These interact with each other and where some interactions may be complementary, others might interfere with each other and lead to worse performance. So the problem statement that we're trying to solve is:
Given a set of optimization flags $N = f o_1, o_2, ..., o_n g$ find the subset $s \quad N$ which leads to the biggest performance improvement.

As the ground set of program optimization has grown,the number of all possible subsets is 2^n and therefore growing exponentially. This makes the computation of the best possible subset a non-trivial problem to solve that has become an ongoing subject of research. Determining the best compiler optimizations, also known as compiler autotuning [3] is not a new problem and has been known since the late 90s [5]. As such, previous approaches to solve this problem include iterative methods [5] or machine learning [4].

More recently [10] uses a linear regression approach to find Fourier-sparse approximations of poset functions mapping combinations and orderings of compiler flags to runtimes.

0.1.2 Contributions

This thesis introduces a PyPi Python library, that composes several Algorithms introduced in various publications into an intuitive library, to provide a basis on which to perform further research and experiments. Specifically this thesis focuses on the algorithms for powersets. The library provides the following functionalities:

- Fast Fourier transform for bases 3, 4 and 5 (WHT)
- Sparse Fourier transform for bases 3, 4 and a weighted variation of 3
- Greedy minimization / maximization for set functions
- Exact minimization / maximization for bases 3 & 4
- Shapley Values for bases 3, 4 and 5
- Various plotting utilities

We provide unittests, documentation and installation guides for all functionalities. We then demonstrate its usability on the example of compiler flag optimizations.

0.2 Signal processing with set functions

The following chapter summarises the theoretical foundation for set function Fourier analysis as presented in [12].

0.2.1 Set functions

Over a finite set N = $f o_1, o_2, ...o_n g$ of size n a set function s is defined as a mapping from the powerset of N (denoted with 2^N) to the domain of real numbers R

$$s : 2^N \; ! \; \mathrm{R}, A \; 7! \; s(A). \tag{1}$$

In other words a set function s relates each subset of N to a real number.
By imposing an order on the subsets of N we can uniquely identify a set function s with a vector **s** = $s(A)_{A \;\; N}$ of dimension 2^N which we will call the spectrum of s. Here we use the lexicographic Ordering on the set of indicator vectors representing subsets. For example, the set of lexicographically ordered indicator vectors representing a ground set of size 3 would then be:

$$f(0,0,0),(1,0,0),(0,1,0),(1,1,0),(0,0,1),(1,0,1),(0,1,1),(1,1,1)g.$$

0.2.2 Shifts

In one-dimensional time and in two-dimensional signal processing a shift operator E^a (also called a translation operator) is defined as the operator that when applied to a function returns its translation.

$$E^a p(x) = p(x + a) \tag{2}$$

[12] generalizes that concept to set functions and defines 5 different shift operators on set functions, each leading to a different discrete-set signal processing (DSSP) model. In this thesis we will focus on the models 3, 4 and 5, where model 5 is equivalent to the Walsh-Hadamard transform. For those the shift operators are defined as

$$\begin{aligned} (model\ 3) &: E^Q s(A) = s(A \,n\, Q) \\ (model\ 4) &: E^Q s(A) = s(A \;[\; Q) \\ (model\ 5) &: E^Q s(A) = s(A \,n\, Q \;[\; Q \,n\, A) \end{aligned} \tag{3}$$

where $Q, A \quad N$.

A shift by a single element can be represented by a matrix. We denote the shift matrix with:

$$\begin{aligned}
(model\ 3) &: f(o_i) = I_{2^n\ i} \begin{matrix} 1 & 0 \\ 1 & 0 \end{matrix} I_{2^i\ 1} \\
(model\ 4) &: f(o_i) = I_{2^n\ i} \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} I_{2^i\ 1} \\
(model\ 5) &: f(o_i) = I_{2^n\ i} \begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix} I_{2^i\ 1}
\end{aligned} \tag{4}$$

0.2.3 Filters

A linear combination of shifts is called a filter and is itself a set function, relating subsets to linear coefficients

$$h = \sum_{Q\ N} h_Q E^Q \tag{5}$$

An important property of filters is that they are shift equivariant. This means that no matter whether you shift first and then apply a filter or the other way around it will not affect the result. With those filters we can form the DSSP equivalent of convolutions:

$$\begin{aligned}
(model\ 3) &: (\mathbf{h} \quad \mathbf{s})(A) = \sum_{Q\ N} h(Q)s(A \cap Q) \\
(model\ 4) &: (\mathbf{h} \quad \mathbf{s})(A) = \sum_{Q\ N} h(Q)s(A \setminus Q) \\
(model\ 5) &: (\mathbf{h} \quad \mathbf{s})(A) = \sum_{Q\ N} h(Q)s(A \cap Q \setminus Q \cap A)
\end{aligned} \tag{6}$$

A filter can then again be represented by a matrix of form:

$$f(h) = \sum_{X\ N} h(X) f(X) \tag{7}$$

0.2.4 Fourier Transformation

The key idea behind the definition of the set Fourier transformation is that it jointly diagonalizes all filters. In matrix representation this means, that $Ff(h)F^{-1}$ is a diagonal matrix, where every entry is a linear coefficient $h(X)$ for $X \subseteq N$, for every filter matrix $f(h)$, where F is the matrix representation of the Fourier transform. To do that, it's enough to diagonalize every shift matrix $f(o_i)$, as matrix multiplication is distributive. F is then of form $T^{\otimes n}$, and T diagonalizes $f(o_i)$. For our models this is true for:

$$\begin{aligned}
(model\ 3) &: T = \begin{matrix} 1 & 0 \\ 1 & 1 \end{matrix} \\
(model\ 4) &: T = \begin{matrix} 0 & 1 \\ 1 & 1 \end{matrix} \\
(model\ 5) &: T = \frac{1}{2}\begin{matrix} 1 & 1 \\ 1 & -1 \end{matrix}
\end{aligned} \tag{8}$$

Note: These are not the only matrices T that work, but they are the ones implemented in the library, so going forward these are the definitions that will be used. Alternatively F can also be represented by a formula that calculates every entry.

$$\begin{aligned}
(model\ 3) &: F = [(-1)^{|A \cap B|} i_{A \subseteq B}]_{A,B \subseteq N} \\
(model\ 4) &: F = [(-1)^{|A \setminus B|} i_{A \subseteq [B=N]}]_{A,B \subseteq N} \\
(model\ 5) &: F = \left(\frac{1}{2}\right)^{n} [(-1)^{|A \setminus B|}]_{A,B \subseteq N}
\end{aligned} \tag{9}$$

Where i_c is the indicator variable for condition c.

$$i_c = \begin{cases} 1, & if\ c\ is\ true \\ 0, & otherwise \end{cases} \tag{10}$$

And $A, B \subseteq N$ are the column and row indices of F, where F is indexed in lexicographical ordering.
The spectrum of $\mathbf{s}$ is denoted as $\hat{\mathbf{s}} = F\mathbf{s}$ and is also a set function and can be written as:

$$\begin{aligned}
(model\ 3) &: \hat{s}(B) = \sum_{A \subseteq N, B \subseteq A} (-1)^{|A \cap B|} s(A) \\
(model\ 4) &: \hat{s}(B) = \sum_{A \subseteq N, A \subseteq [B=N]} (-1)^{|A \setminus B|} s(A) \\
(model\ 5) &: \hat{s}(B) = \frac{1}{2^n} \sum_{A \subseteq N} (-1)^{|A \setminus B|} s(A)
\end{aligned} \tag{11}$$

0.2.5 Convolution Theorem

As proven in [12] we can then conclude for convolutions with filter h

Theorem 1

$$\widehat{\mathbf{h} * \mathbf{s}} = \hat{\mathbf{h}} \cdot \hat{\mathbf{s}} \tag{12}$$

0.3 Algorithms and Functionalities

In this chapter we will introduce the algorithms and the different functionalities that make up the provided Python library. It summarises the content from [12], [16] and [14]. We will not cover all details and will refer to these papers for proofs and more information. For each algorithm we will first cover the theoretical background and then provide a short example of how the library can be used to apply that theory in praxis.

0.3.1 Fast Fourier Transforms

The fast Fourier transform is an algorithm that exploits the special recursive structure $F = T^{\otimes n}$ of the Fourier matrix to calculate the spectrum $\hat{\mathbf{s}}$ for a given full signal vector $\mathbf{s}$, as was done in [7] for the fast Walsh-Hadamard transform algorithm. It uses a divide-and-conquer approach, where in each step of the algorithm it halves the size of the transformation matrix. Below is an example of the FWHT for a signal of length 4 (i.e ground set size 2)

$$F\mathbf{s} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} s_1 + s_2 \\ s_3 + s_4 \\ s_1 - s_2 \\ s_3 - s_4 \end{bmatrix} \tag{13}$$

$$= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} s_1 + s_2 \\ s_1 - s_2 \\ s_3 + s_4 \\ s_3 - s_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} s_1 + s_2 \\ s_1 - s_2 \\ s_3 + s_4 \\ s_3 - s_4 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} s_1 + s_2 \\ s_1 - s_2 \\ s_3 + s_4 \\ s_3 - s_4 \end{bmatrix} = \begin{bmatrix} s_1 + s_2 + s_3 + s_4 \\ s_1 + s_2 - s_3 - s_4 \\ s_1 - s_2 + s_3 - s_4 \\ s_1 - s_2 - s_3 + s_4 \end{bmatrix} \tag{14}$$

The Fourier transform algorithm for the other models follow the same pattern, but use their respective Fourier transform matrices, as defined in (8).

Algorithm 1 Fast Walsh-Hadamard Transform

```
h = 1
while h < len(signal) do
    for i = 0; i < len(signal), i = i + 2h do
        for j = i, j < i + h, j = j + 1 do
            x = signal[i]
            y = signal[j]
            signal[i] = x + y
            signal[j] = x - y
    signal = signal / 2
    h = 2h
```

Calculating a naive implementation of the Fourier transform for a signal of

length 2^n would have a computational complexity of $O(2^{2n})$. With the fast Fourier transform algorithms we're able to reduce the complexity to $O(n2^n)$. As mentioned in the introductory chapter, the computation of the complete signal of a set of compiler optimizations would take 2^N queries of the set function,which is not feasible for modern compilers, where the number of individual compiler flags can often be three-digits. For example gcc's optimization level -O3 consists of around 100 compiler flags. While this does not help directly with determining the best possible subset, it allows us to compute the full spectrum of small enough set functions. In practice this is true for ground set sizes 29. We can use this to analyze smaller set functions and potentially apply our findings to bigger set functions.

Application

```
#import necessary module
from dsft import setfunctions

#example spectrum of a setfunction with n = 4
signal = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
#create a callable setfunction representation of that
    spectrum
setfunction = setfunctions.WrapSignal(signal)
#apply fast fourier transform for the desired model
fft3 = setfunction.transform_fast(model ='3')
fft4 = setfunction.transform_fast(model = '4')
fwht = setfunction.transform_fast(model = '5')
#print spectrum of transformed setfunction
print(fft3.coefs)
print(fft4.coefs)
print(fwht.coefs)
```

Listing 1: Example usage of fast Fourier transform.

Plotting the coefficients of these transforms against the cardinalities of their respective frequencies (Fig. 1) gives us insight as to why calculating the Fourier transformation can be useful for working with set functions. In this small example we see that the information contained in the original set function was distributed across all cardinalities, while the Fourier transformations compresses the information into only a few cardinalities. Therefore we can completely reconstruct the original set function when we know all non-zero Fourier coefficients.

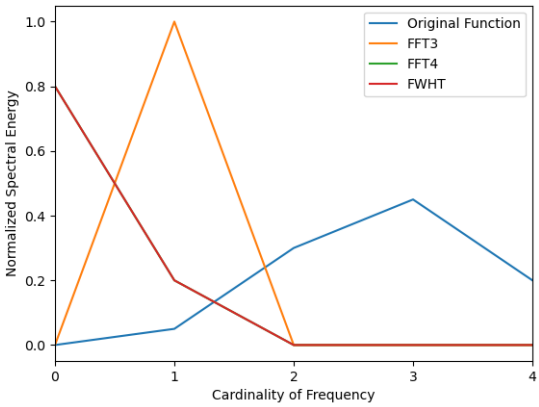

Figure 0.1: Normalized average coe cient plotted agianst the cardinality of their correlating frequency for the example from Listing 1 (FFT4 coincides with the FWHT in this case)

0.3.2 Sparse Fourier Transform

Ideally we would like to be able to compute the Fourier transformation of a set function without having to evaluate it for all subsets first, like we need to do for the fast transformation presented earlier. The sparse Fourier transform algorithm presented in [16] provides a way to do exactly that for Fourier-sparse set functions, while only imposing mild conditions on the Fourier coefficients.

Fourier Sparsity and Support

Definition 0.3.1 *We call the set*

$$supp(\hat{s}) = \{B : \hat{s}(B) \neq 0\} \tag{15}$$

the support of set function $\hat{s}$

Definition 0.3.2 *A set function s is called k-Fourier-sparse if*

$$supp(\hat{s}) = B_1, ..., B_k \tag{16}$$

where we assume that k is significantly smaller than 2^N

Therefore it's sufficient to find the non-zero Fourier coefficients and their respective support to exactly learn a set function. Interestingly our different Fourier transform models each lead to a different sparsities. This means that a function can be sparse in one model, but not for others.

Sparse FT with known support

Under the assumption that the Fourier support $supp(\hat{s})$ or a small superset $B \supseteq supp(\hat{s})$ is known, the problem of learning the set function s can be simplified to finding a set of frequencies $A \subseteq 2^n$ such that

$$\mathbf{s}_A = F_{AB}^{-1} \hat{\mathbf{s}}_B \tag{17}$$

has a unique solution, where $\mathbf{s}_A = (s(A))_{A \in A}$ is the vector of s queried at every element of A and F_{AB}^{-1} is the F^{-1} indexed by the elements of A, B Then according to [12]:

Theorem 2 *Let s be k-Fourier-sparse with* $supp(\hat{s}) = \{B_1, ..., B_k\} = B$. *Let* $A = \{N \oplus B_1, ..., N \oplus B_k\}$. *Then* F_{AB}^{-1} *is invertible and s can be perfectly reconstructed from the queries* $\mathbf{s}_A$

Therefore the problem of computing the Fourier coefficients can be reduced to finding the associated support.

Sparse FT with unknown support

Finding the support of a set funtion s is not a trivial task, and doing it naively would mean checking all 2^n possible frequencies. The method presented in [16] introduces the notion of restricted set functions, which are set functions defined over a subset $M \subseteq N$

Definition 0.3.3 *For any* $M \subseteq N$ *we denote the restricted set function with*

$$s|_{2^M} : 2^M \to \mathbb{R}, A \mapsto s(A) \tag{18}$$

As defined and proven in [16] we can then formulate the Fourier transform for model 4 as

$$s|_{2^M}^{\wedge}(B) = \sum_{A \subseteq N \oplus M} \hat{s}(A \oplus B) \tag{19}$$

Under the assumption that there are no cancellations in the summation of (19) we can deduce that every restricted set function can be related to a less restricted set function.

$$s|_{2^M}^{\wedge}(B) = s|_{2^{M \setminus \{x\}}}(B) + s|_{2^{M \setminus \{x\}}}(B \oplus \{x\}) \tag{20}$$

for any $x \in N \oplus M$. This implies that whenever $s|_{2^M}^{\wedge}(B)$ is zero then $s|_{2^{M \setminus \{x\}}}(B)$ and $s|_{2^{M \setminus \{x\}}}(B \oplus \{x\})$ will be zero too. Therefore we have

$$B = \bigcup_{B \in supp(s|_{2^M}^{\wedge}(B))} \{B, B \oplus \{x\}\} \tag{21}$$

with $supp(s|_{2^{M \setminus \{x\}}}) \subseteq B$ which allows us to use Theorem 1 to calculate $s|_{2^{M \setminus \{x\}}}$. So now we can compute the Fourier coefficients of a restricted set function $s|_{2^{M \setminus \{x\}}}$, if we have the Fourier coefficients of $s|_{2^M}^{\wedge}$. This allows us

to build a chain starting with $M_0 = ?$ and $s\#_{2^?}(?) = s(?)$, where in each iteration we increase the set M_i by adding the next element then compute the superset B for $supp(s\#_{2^{M_i}})$ and apply Theorem 1 to calculate the coefficients of it. After repeating this n-times we will have added all elements of N and therefore we have computed the coefficients of $s\#_{2^{M_i}} = \mathbf{b}$

Algorithm 2 Sparse Set Fourier Transform (SSFT)

$M_0 \quad ?$
$s\#_{2^?}(?) \quad s(?)$
for $i = 1, ..., n$ **do**
  $M_i \quad M_{i-1} \cup \{x_i\}$
  $A \quad ?, B \quad ?$
  for $B \in supp(s\#_{2^{M_{i-1}}})$ **do**
    $B \quad B \cup \{B, B \cup \{x_i\}\}$
    $A \quad A \cup \{M_i \setminus B, M_i \setminus (B \cup \{x_i\})\}$
  $\mathbf{s}_A \quad (s(A))_{A \in \mathcal{A}}$
  $\mathbf{x} \quad solve\ \mathbf{s}_A = F^{-1}_{\mathcal{A}\mathcal{B}} x\ for\ x$
  for $B \in \mathcal{B}\ with\ \mathbf{x}_B \neq 0$ **do**
    $s\#_{2^{M_i}}(B) \quad \mathbf{x}_B$
return $s\#_{2^{M_n}}$

This algorithm is guaranteed to compute the Fourier transform of a k-sparse set function, under conditions that guarantee that no cancellation occurs. This is the case for set functions with Fourier coefficients sampled from independent continuous probability distributions. Compared to the fast Fourier transform presented above, which needs $O(2^n)$ queries and $O(n2^n)$ operations, the sparse algorithm manages to solve the problem with a maximum of $O(nk \quad k \log k)$ queries and $O(nk^2)$ operations.

Filtering

As seen before the **SSFT** only works on a limited set of set functions. But some important families of set functions like graph-cuts are not included. To extend the set of set functions that work, we can utilize the convolution theorem, where we find a random filter h so that SSFT works with probability one for the convolution $s \quad h$. (12) states that we can then recover the coefficients by dividing pointwise with the frequency response of h

$$\mathbf{b} = \mathbf{h} \cdot \mathbf{s} / \bar{h} \tag{22}$$

To minimize the added overhead of the filtering , we choose h to be a one-hop-filter, meaning a filter where $h(B) = 0$ for $|B| \quad 1$ This extension results in a new complexity of $O(n^2k \quad nk \log k)$ queries and $O(n^2k + nk^2)$ operations.

Other Models

We've seen how we defined the SSFT for model 4 above, and consequentially we can define it for all other Fourier bases presented in [12]. Implemented in the library are the SSFT algorithms for model 3, 4 and a weighted variation of 3. These adaptations for model 3 and its weighted variation are from [6] For model 3, the equation equivalent to (18) is

$$s\#_{NnM[2^M}(B) = \sum_{A \ NnM} (-1)^{|A \cap B|} s(A \cap B) \tag{23}$$

Analogous we can then construct the chain of subproblems to solve. We start with $s\#_{N[2^?}$ and in each step we add the next element, compute the superset and apply Rheorem 1 to get the coefficients. After n steps we end up with the coefficients of the original set function. The weighted variation of model 3 is based on the Fourier transform matrix

$$(model\ 3) : T = \begin{bmatrix} 1 & 0 \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} \end{bmatrix} \tag{24}$$

and it's equivalent to (18) is

$$s\#_{NnM[2^M}(B) = 2^{-|NnM|} \sum_{A \ NnM} (\sqrt{3})^{|A \cap B|} s(A \cap B) \tag{25}$$

from there we proceed like we do for model 3

Application

```
#import necessary modules
from dsft import setfunctions
from dsft import datasets
from dsft import plotting

#load compiler optimization setfunction for benchmark program
    (susan_c)
comp_opt = datasets.load_bench(bench = 'susan_c10')
#create sparse estimations of the fourier transforms
comp_opt_ssft3 = comp_opt.transform_sparse(model = '3',eps =
    1e-3,flag_general = False)
comp_opt_ssftW3 = comp_opt.transform_sparse(model = 'W3',eps
    = 1e-3,flag_general = False)
comp_opt_ssft4 = comp_opt.transform_sparse(model = '4',eps =
    1e-3,flag_general = False)
#plot spectral energy
plotting.plot_spectral_energy([comp_opt_ssft3,comp_opt_ssftW3
    ,comp_opt_ssft4],['SSFT3','SSFTW3','SSFT4'],n=10)
```

Listing 2: Example usage of sparse Fourier transform.

Applying the sparse Fourier transform works in a very similar way to applying the fast transform. With the difference, that the set function we apply

it to can be a queryable function, where we do not have to know all 2^n set function evaluations. Additionally we can create sparse function approximations by setting the eps parameter. Higher values for eps lead to a more sparse function, as it means the threshold below which we consider a coefficient to be zero will be set higher. We have to take into consideration that this is not the intended use of the algorithm and that it is not guaranteed to work when the set function is not exactly sparse and when some of the coefficients cancel out. To assure that we get correct results we should, if possible, strive for an absolute error of the Fourier coefficients of 1.

0.3.3 Minimization

A common goal across many applications of our previously defined set-function theory, is the minimization or maximization of a set function. We denote it with

$$\underset{a}{\operatorname{argmax}}\, s(a) \qquad \underset{a}{\operatorname{argmin}}\, s(a). \tag{26}$$

This also applies to the problem of finding the optimal subset of compiler optimizations that we're discussing in this thesis. To this end our Python library contains 2 different algorithms that can be used to compute the indicator vector that maximizes or minimizes a certain set function.

Greedy Algorithm

The first method implemented in our library is the greedy algorithm. It starts out with the empty set $s = ?$ and then in each iteration searches for the element $s_i \; 2 \; s$ which when added to s will lead to the biggest increase in the set function output. It repeats this process until a specified cardinality threshold is reached, or until adding more elements will not further increase the output.

Algorithm 3 Greedy set function Maximization

```
1: sf    input setfunction
2: s     ?
3: for c = 1,...,card_max do
4:   for s_i 2 s do
5:     value_max = sf(s)
6:     if sf(s [ s_i) > value_max then
7:       element_max = s_i
8:       value_max = sf(s [ s_i)
9:     else if sf(s [ s_i) = value_max then
10:      choose randomly whether to choose i
11:  s    s [ element_max
12: return s
```

This algorithm allows us to find a solution in $O(n^2)$ queries of the set function which is a significant reduction from the $O(2^n)$ queries we would need if we were to check all subsets. Unfortunately this algorithm doesn't guarantee that for all functions the best possible solution will be found. There are situations, where choosing a certain element, will lead to missing out on finding an optimal solution that does not contain that element. However for the class of submodular functions it is guaranteed to find a solution that statisfies

$$max : s(\bar{x}) \quad (1 \quad e)s(x_{opt}), \qquad min : s(\bar{x}) \quad (1 \quad e)s(x_{opt}), \tag{27}$$

where $\bar{x}$ is the found solution and x_{opt} would be the best possible solution. [8]

```
#import necessary modules
from dsft import setfunctions
from dsft import datasets
#load callable compiler optimization setfunction for
    benchmark program (susan_c)
comp_opt = datasets.load_compiller_opt(benchmark = 'susan_c',
    n=10)
#perform greedy minimization on original setfunction
minimizer,value = comp_opt.minimize_greedy(n=10,max_card =
    10)
#Sparse Fourier transform algorithms
comp_opt_ssft3 = comp_opt.transform_sparse(model = '3')
#perform greedy minimization on sparse transform
minimizer_s3,value_s3 = comp_opt_ssft3.minimize_greedy(n=10,
    max_card = 10)
```

Listing 3: Example usage of the greedy minimization.

MIP based Algorithm

The second method is the implementation of Theorem 1 from [14], where the maximization problem is reformulated as a mixed integer program (MIP) that utilizes Fourier sparsity. They redefine the problem of maximization

$$\underset{x,b2f0,1g^k}{\operatorname{argmax}} hb, ai \qquad \underset{x,b2f0,1g^k}{\operatorname{argmin}} hb, ai \tag{28}$$

where hx, yi denotes the euclidian scalar product, as a set of constraints that are specific to the chosen model:

$$(model\ 3): \begin{array}{rcl} a_i & 1 & y^T(1 \quad x) \\ a_i & 1 & y^T(1 \quad x) + Cb_i \\ a_i & C & (1 \quad b_i) \end{array} \tag{29}$$

$$(model\ 4): \begin{array}{rcl} a_i & 1 & y^T x \\ a_i & 1 & y^T x + Cb_i \\ a_i & C & (1 \quad b_i) \end{array} \tag{30}$$

$$(model\ 5): \begin{array}{rcl} a_i & = & 2b_i + 1 \\ b_i & = & y^T x \quad 2g_i \\ g_i & 2 & Z^k \end{array} \tag{31}$$

These constraints can then be fed into a specialized solver such as scipopt.

```
#import necessary modules
from dsft import setfunctions
from dsft import datasets
#load full compiler optimization setfunction for benchmark
    program (bitcount)
comp_opt = datasets.load_bench_bitcount10()
#Sparse Fourier transform algorithms
comp_opt_ssft3 = comp_opt.transform_sparse(model = '3',
    flag_general = False)
#perform MIP minimization on sparse transform
minimizer_s3 , value_s3 = comp_opt_ssft3 . minimize_MIP()
```

Listing 4: Example usage of the MIP-based minimization.

0.3.4 Shapley Values

Shapley values introduced by Lloyd Shapley in 1951 [13] is a method to measure the gain per player in a coalition game (set function). In our case we can use it as a way to measure and compare how much each compiler flag contributes to the speedup. Formally the Shapley value of the i-th player in a coalition game s is defined as

$$j_i(s) = \sum_{A \quad Nnfig} \frac{jAj!(n \quad jAj \quad 1)!}{n!}(s(A \; [\; fig) \quad s(A)) \tag{32}$$

According to [15], for sparse set functions with $supp(s) = \mathbf{B}$ we can write it as

$$j_i(s) = j_i(\sum_{B2\mathrm{B}} s(B) f^B) = \sum_{B2\mathbf{B}} s(B) j_i(f^B) \tag{33}$$

where f^B denotes the B-th fourier base vector (the B-th column of the inverse Fourier transform matrix $F^{\ 1}$. $j(f^B)$ once again depends on the model we choose

$$\begin{aligned}
(model\ 3) : j(f^B) &= \begin{cases} \frac{1}{jBj}, & if\ i\ 2\ \mathbf{B} \\ 0, & otherwise \end{cases} \\
(model\ 4) : j(f^B) &= \begin{cases} \frac{1}{jBj}, & if\ i\ 2\ \mathbf{B} \\ 0, & otherwise \end{cases} \\
(model\ 5) : j(f^B) &= \begin{cases} \frac{(\ 1)^{jBj} \ 1}{jBj}, & if\ i\ 2\ \mathbf{B} \\ 0, & otherwise \end{cases}
\end{aligned} \tag{34}$$

```
#import necessary modules
from dsft import setfunctions
from dsft import datasets
#load full compiler optimization setfunction for benchmark
    program (bitcount)
comp_opt = datasets.load_bench_bitcount10()
#Sparse Fourier transform algorithms
comp_opt_ssft3 = comp_opt.transform_sparse(model = '3',
    flag_general = False)
#calculate shapley_values
shapley_values = comp_opt_ssft3.shapley_values()
#print shapley_values
print(shapley_values)
#OUTPUT:
#[ 1.17470218e-04  2.47338553e-05  4.76721742e-05  3.16573169
    e-05
#  5.26569980e-05 -2.59871229e-04 -5.47762424e-04 -2.68371264
    e-03
# -1.14258840e-03 -6.19581143e-03]
```

Listing 5: Example usage of the shapley value function.

0.4 Implementation

In this chapter we will present our PyPi package setFTs and introduce the important modules and classes. For more in-depth information, we also created a full documentation for the modules setfunctions and plotting, which can be found at: `https://ebners.github.io/setFTs_docs/`

0.4.1 Package Structure

The library contains various Python modules with implementations of the algorithms and functionality mentioned above.

- **set functions**:The module that a user will mainly interact with. Contains classes for set function representations and their Fourier transformed variations with functions to calculate maximization, shapley values and the spectral energy respectively.
- **plotting:** Functions to help visualize the generated data
- **transformations:** Provides classes that instantiate Fourier transform objects, that allow to perform both fast and sparse Fourier transformation on a set function object.
- **minmax:** module for MIP-based minimization or maximization.
- **datasets:** Loader functions for some included datasets for set functions
- **utils:** Contains utility functions for generating full sets of indicator values.

0.4.2 setfunctions module

The library is designed in a way that a user will usually only have to interact with the setfunctions module. This was chosen to make the library easy to understand and apply to new set funtions. Here we will cover the basics of this module. That module consists of a class called 'set function' and multiple subclasses:

- **WrapSignal**: Used for instantiating a set function representation, when all setfunction evaluations are already known. Class instantiation works with a list of 2^n float values that represent the set function evaluations in lexicographical ordering.
  (e.g. sf = setfunctions.WrapSignal([0,1,2,3,4,5,6,7])
- **WrapSetFunction**:Used for instantiating a set function representation for queryable set functions. Class instantiation needs a parameter s, that is a callable function that takes a one-dimensional numpy array as an input and returns a float value, and the groundset size n.
  (e.g. sf = setfunctions.WrapSetFunction(examplesf,n)
- **SparseDSFTFunction**: Allows the instantiation of a sparse Fourier transformed set function. An object of this class can be created directly with its non-zero Fourier coefficients and their respective frequencies or by applying either of the transform functions (fast or sparse) on either of the wrapper functions above.
  (e.g ssft3 = setfunctions.SparseDSFT3Function(freqs,coefs)
  or ssft3 = sf.transform_sparse(model = '3',eps = 1e-3)

0.4.3 Plotting utilities

The second module that might be of interest for users of this library is the plotting module, which provides utility functions to create plots for setfunctions. The plotting functions are all based on the well-documented plotting library matplotlib.

- **plot_freq_card**: Creates a histogram or line plot of the number of frequencies of each cardinality in a setfunction. (Figure 0.2)

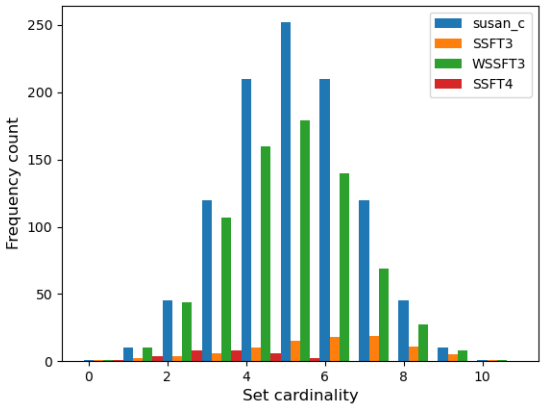

Figure 0.2

- **plot_spectral_energy**: Creates a histogram or line plot showing the average normalized coefficients for each cardinality. (Figure 0.3)

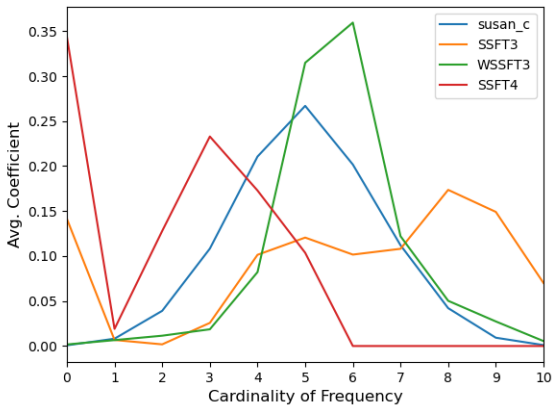

Figure 0.3

- **plot_min_greedy/plot_max_greedy**: Line plot of the smallest / largest set function value found by the greedy minimization or respectively maximization algorithm when restricted to a maximal cardinality. (Figure 0.4)

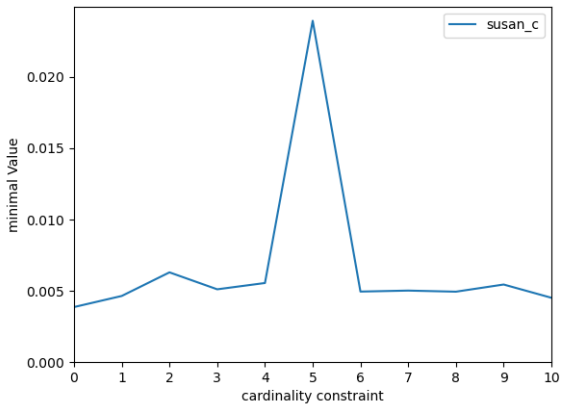

Figure 0.4

- **plot_max_mip/plot_min_mip**: Line plot of the smallest / largest set function value found by the MIP-based minimization or respectively maximization algorithm when restricted to a maximal cardinality. (Figure 0.5)

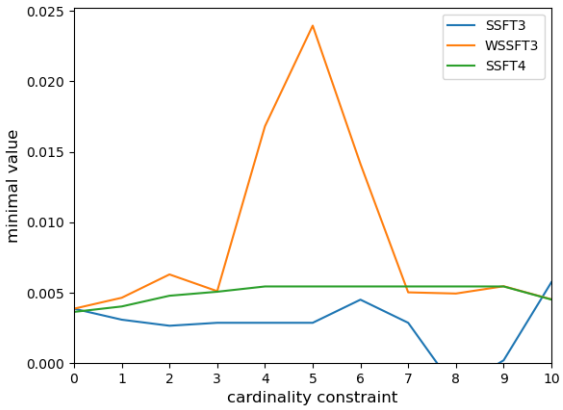

Figure 0.5

- **plot_scatter**: Creates a scatter plot showing the distribution of the coefficient size for each cardinality. (Figure 0.6)

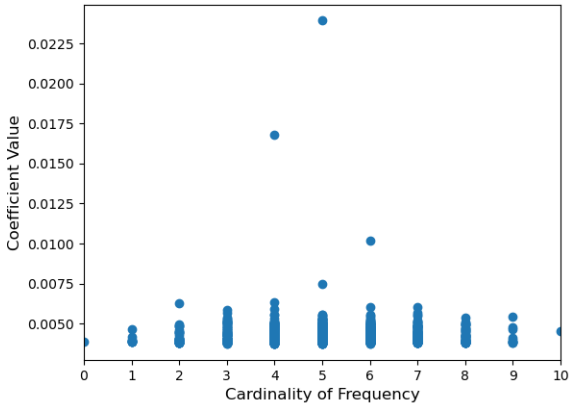

Figure 0.6

- **plot_reconstruction_error**: Creates sparse approximations of a setfunction by varying the eps parameter and plots the reconstruction error. (Figure 0.7)

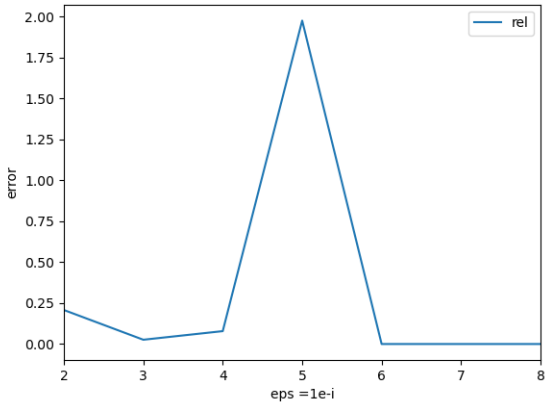

Figure 0.7

- **plot_reconstruction_error_biggest_coefs** Calculates the sparse Fourier transformations and then plots the reconstruction errors when constrained to their k-biggest coefficients. (Figure 0.8)

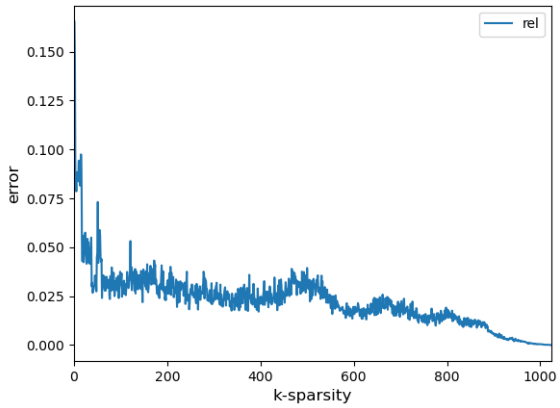

Figure 0.8

- **plot_minimization_found** Functions like plot_reconstruction_error, but performs for each eps value also the MIP minimization algorithm(Figure 0.9)

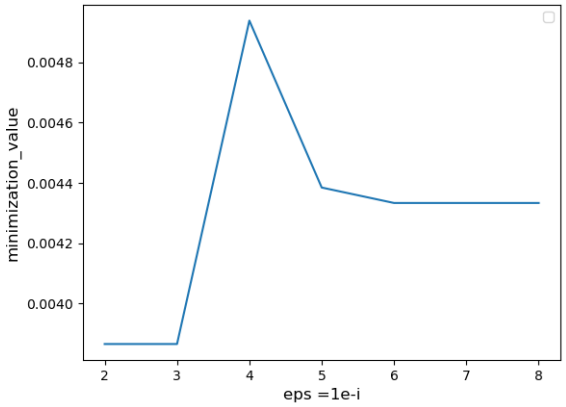

Figure 0.9

- **plot_minimization_found_biggest_coefs** Similar to plot_reconstruction_error_biggest_coefs, but applies a minimization algorithm at each step.(Figure 0.10)

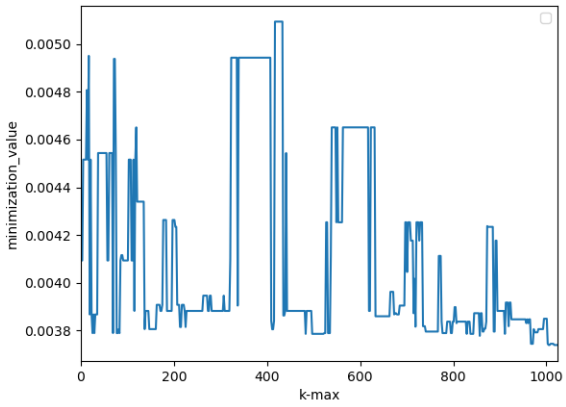

Figure 0.10

0.4.4 Unittests

To ensure that the code works correctly, we wrote unittests for almost all basic functions of our library. For testing our fast and sparse algorithms we test on small known examples whether our Fourier transform computes the correct coefficients, and whether we get the same results by calling the transformed function as we do when calling the original set function on randomly generated signals. We use the same random signals to test whether the minimization found by the MIP-minimizer is the same as the smallest value in the random signal. Shapley values have the property that they are the same for every model presented, which allows us to calculate them for random signals and compare the results of different models for equality.

0.4.5 Installation

setFTs uses the Python library pySCIPOpt for the implementation of the MIP-based minimization algorithm. pySCIPOpt requires a working installation of the SCIP Optimization Suite. The creators of pySCIPOpt recommend using conda as it installs SCIP automatically. And allows the installation of pySCIPOpt in one command:

conda install channel conda forge pyscipopt

The installation of our package works over PyPi and therefore a working installation of pip is needed. The pip command to install setFTs is the following:

pip install setFTs

0.5 Experiment on Compiler Optimizations

0.5.1 Experiment setup

To apply our library to compiler optimizations, we use the framework provided by [10]. It allows us to create a queryable set function that measures the runtime of a specified benchmark program compiled with the compiler flags specified in the query. To produce more stable results it repeats each measurements multiple times, removes outliers and takes the mean of the remaining measurements. We use this to create full datasets of all set function evaluations for 8 different benchmark programs with a collection of 10 compiler flags. The flags used for these examples are:
-foptimize-sibling-calls, -freorder-blocks-algorithm=stc, -ftree-pre, -fschedule-insns2, -fexpensive-optimizations, -fvect-cost-model, -ffinite-loops, -floop-unroll-and-jam, -flra-remat, -fvect-cost-model=dynamic
which are 10 of the around 100 compiler flags of optimization level -O3. We limit ourselves to 10 flags in this case, because we create a full set of set function evaluations to be able to compare the solution of our algorithms with the best possible solution that could be obtained by the naive approach. The calculation of bigger sets would take a relatively long time and the smaller sets are enough for demonstration purposes. In general the set functions obtained by querying a benchmark program are not Fourier sparse and it will therefore take 2^n queries to calculate their Fourier transforms even with our sparse algorithm. As mentioned before, the querying of a large amount of subsets is time intensive, and therefore we would like a way to calculate it using a minimal amount of queries.
We know that we can create a sparse approximation of the Fourier transform by setting the eps parameter of the sparse algorithm higher. While it is not advised to do this, as this can lead to inconsistent and wrong results in some cases, we will do it here for experimentation purposes.
We then create such a sparse approximation for each benchmark program and run the MIP-based minimization algorithm on them. This produces an approximation of the minimizing subset, but will in most cases not be the optimal solution.

0.5.2 Results

Table 0.1 in the addendum shows the resulting data of this experiment. For all programs we tested an eps value of $eps = 1e-4$ and increased it to $eps = 1e-3$ for programs where the number of queries for the sparse approximation was still relatively high.
Both models show a similar performance in these examples, where the minimization found is relatively close to the optimal solution for a majority of programs. But unfortunately for some benchmark programs the results are above the average set function value and therefore too far off the minimal value to be considered a valid result. The amount of queries required is significantly reduced in both. Compared to the minimal value possible, the

average relative error of the minimal value of the approximated set function is 0.030631665 for model 3 and 0.029863149 for model 4. We repeated this experiment with the filtered SSFT algorithm, which resulted in better results, but the reproduction of these values was inconsistent. To counter this inconsistency we tried repeating the experiment multiple times for each program. This resulted in some cases in more queries done than would be needed in the naive approach. These additional results can be found in the addendum under table 0.2.
The results also varied strongly depending on the eps parameter chosen. By plotting the reconstruction error for different eps values (Figure 0.9), we can see why that is the case.

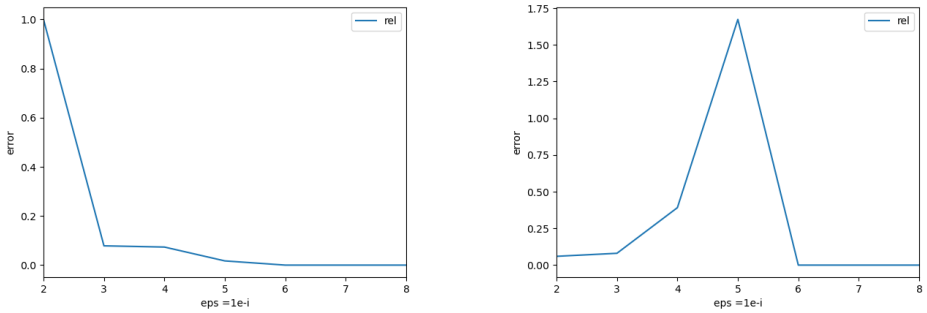

Figure 0.11: Comparison of the reconstruction error of sparse approximations with the SSFT3 (left) and the SSFT3+ (right)

0.5.3 Shapley values

Another experiment we performed was the calculation of the Shapley values for each of the benchmark programs to determine which single individual compiler flag had the largest impact on the runtime. To be able to compare the Shapley values between different benchmark programs, we normalize by dividing each value by the sum of all Shapley values of that program. These normalized Values can be found in table 0.3 of the addendum. Of the 10 compiler flags we consider in this small example,"-ftree-pre", "-floop-unroll-and-jam" and "-freorder-blocks-algorithm=stc" seem to be making the biggest impact on average. According to [1] "-ftree-pre" is a flag that enables partial redundancy elimination on trees, which means that it eliminates redundancies for expressions that are redundant in some paths of the programm but might not be redundant in other paths.
And "-floop-unroll-and-jam" is responsible for loop unrolling and jaming, where for nested loops the outer loop will be unrolled and the resulting inner loops will be fused. This allows more reusage in the loop.
"-freorder-blocks-algorithm=stc" sets the reordering algorithm for basic code blocks to software trace cache which groups code that is often executed together.

0.6 Future Work

0.6.1 Extending the Library

There exist more related algorithms that could be implemented in future iterations of this library. For example, we could implement our sparse Fourier transform algorithm for model 5, where we would need to redefine the subproblem as is discussed in addendum E of [16]. Another algorithm that would be worth implementing is the Sparse Walsh Hadamard transform presented in [2]. By applying ideas from compressive sensing over finite fields it provides a way to calculate the WHT in $O(kn \log^2 k \log n \log d)$ operations and $O(kd \log n)$ queries, if we know that all frequencies that make up the support $supp(\hat{b})$ of the Fourier transform $\hat{b}$ are of low order, meaning that their cardinality is lower than some threshold d.
Another idea how the library could be expanded is by not only restricting ourselves to set functions, but extending it to also include functionalities over partially ordered sets, specifically meet/join lattices aswell. [11] introduces the theoretical foundation for that. Partially ordered sets have the added benefit, that there is an imposed partial order over the set which would enable us to also take different permutations of the subset into account.

0.6.2 Research on Compiler Optimizations

The experiment we conducted highlighted some important limitations of this approach. One of them is the problem of creating consistent Fourier sparse approximations. As discussed in the results section, the estimations we created with the sparse algorithm worked a lot better for some programs than for others. A way to possibly refine the results of our approach would be to fine tune the choice of our eps parameter and allow for values that are more fine-grained than $eps = 1e\ \ i$ for $i \ 2$ R, because during testing it occured multiple times that for a given program the amount of coefficients obtained was too high, when using one value for eps, but too low when using the next higher value. We can also consider a different approach to create approximations. Current research on optimizing compiler flags as in [10] is directing its focus towards using machine learning to learn and minimize estimations of set functions or in this case even poset functions. Further, the experiments conducted here were limited to a selection of 10 compiler optimizations. This is just a fraction of all that would be available, so further research needs to be done on a wider subset of optimizations.

0.7 Addendum

FT3

Benchmark Program	Min. Value	Avg. Value	Min. Est FT3	Queries Est FT3	rel. Error Est Ft3
bitcount	0.04	0.047052055	0.040056	289	0.0055555
susan_c	0.00371795	0.00428696	0.004938	126	0.03874447
susan_e	0.006791	0.007092	0.006831	19	0.0060374
susan_s	0.056313	0.058198	0.057313	661	0.01308756
susan_s_eps3	0.056313	0.058198	0.05745	17	0.02019072
bzip2d	0.04875	0.053067	0.050875	750	0.01602051
bzip2d_eps3	0.04875	0.053067	0.05535	347	0.02947692
bzip2e	0.0805	0.083245	0.083312	765	0.04658385
bzip2e_eps3	0.0805	0.083245	0.082812	11	0.0287205
jpeg_c	0.0023	0.002677	0.002431	94	0.0226087
jpeg_d	0.001028	0.001284	0.001425	55	0.10992218

FT4

Benchmark Program	Min. Value	Avg. Value	Min. Est FT4	Queries Est FT4	rel. Error Est Ft4
bitcount	0.04	0.047052	0.040222	88	0.00555
susan_c	0.003718	0.004287	0.004067	42	0.09386767
susan_e	0.006791	0.007092	0.006995	71	0.03003976
susan_s	0.056313	0.058198	0.058562	752	0.03993749
susan_s_eps3	0.056313	0.058198	0.05745	38	0.02019072
bzip2d	0.04875	0.053067	0.051281	944	0.05191795
bzip2d_eps3	0.04875	0.053067	0.052775	574	0.0825641
bzip2e	0.0805	0.083245	0.0849	912	0.05465839
bzip2e_eps3	0.0805	0.083245	0.0825	63	0.02484472
jpeg_c	0.0023	0.002677	0.002371	20	0.03086957
jpeg_d	0.001028	0.001284	0.001255	217	0.22081712

Table 0.1: Result of sparse approximation on di erent benchmarks

FT3+

Benchmark Program	Min. Value	Avg. Value	Min. Est FT3	Queries Est FT3	rel. Error Est Ft3
bitcount	0.04	0.047052055	0.04022222	34	0.0055555
susan_c	0.00371795	0.00428696	0.003862	50	0.03874447
susan_e	0.006791	0.007092	0.006832	45	0.0060374
susan_s	0.056313	0.058198	0.05705	469	0.01308756
susan_s_eps2	0.056313	0.058198	0.05745	17	0.02019072
bzip2d	0.04875	0.053067	0.049531	655	0.01602051
bzip2d_eps2	0.04875	0.053067	0.050187	37	0.02947692
bzip2e	0.0805	0.083245	0.08425	248	0.04658385
bzip2e_eps2	0.0805	0.083245	0.082812	49	0.0287205
jpeg_c	0.0023	0.002677	0.002352	71	0.0226087
jpeg_d	0.001028	0.001284	0.001141	12	0.10992218

FT4+

Benchmark Program	Min. Value	Avg. Value	Min. Est FT4	Queries Est FT4	rel. Error Est Ft4
bitcount	0.04	0.047052055	0.04022222	23	0.0055555
susan_c	0.00371795	0.00428696	0.003865	18	0.03955137
susan_e	0.006791	0.007092	0.006939	12	0.02179355
susan_s	0.056313	0.058198	0.056688	362	0.00665921
susan_s_eps2	0.056313	0.058198	0.05675	32	0.0077602
bzip2d	0.04875	0.053067	0.050875	624	0.04358974
bzip2d_eps2	0.04875	0.053067	0.05175	22	0.06153846
bzip2e	0.0805	0.083245	0.0825	243	0.02484472
bzip2e_eps2	0.0805	0.083245	0.082125	34	0.02018634
jpeg_c	0.0023	0.002677	0.002339	25	0.01695652
jpeg_d	0.001028	0.001284	0.001141	12	0.10992218

Table 0.2: Result of sparse approximation with random one hop filtering on different benchmarks

Benchmark Program	”-foptimize-sibling-calls”	”-freorder-blocks-algorithm=stc”	”-ftree-pre”	”-fschedule-insns2	”-fexpensive-optimizations”	”-fvect-cost-model”	”-ffinite-loops”	”-floop-unroll-and-jam”	”-flra-remat”	”-fvect-cost-model =dynamic”
bitc	-0.01	0.00	0.00	0.00	0.00	0.02	0.05	0.25	0.11	0.59
susan_c	0.12	0.01	0.11	0.02	0.17	0.22	0.03	0.12	-0.26	0.45
susan_e	0.13	-0.11	0.39	0.01	0.07	0.12	0.14	-0.02	0.28	-0.01
susan_s	0.18	0.19	0.19	0.04	0.34	0.03	0.01	-0.01	0.03	0.00
bzip2d	0.12	0.32	0.10	-0.01	-0.04	0.01	-0.05	0.13	0.20	0.21
bzip2e	-0.10	0.49	1.54	0.23	-0.38	-0.37	-0.30	1.00	-0.37	-0.75
jpeg_c	-0.10	0.10	0.08	0.05	-0.13	0.02	-0.17	0.05	0.52	0.58
jpeg_d	0.07	0.15	0.24	0.02	0.06	0.08	0.29	0.15	-0.01	-0.05
Average	0.05	0.14	0.33	0.05	0.01	0.02	0.00	0.21	0.06	0.13
Sum	0.46	1.29	2.98	0.41	0.10	0.17	0.00	1.89	0.56	1.14

Table 0.3: Normalized Shapley values of di erent benchmark programs

Bibliography

[1] Accessed: 2022-08-26. url: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[2] Andisheh Amrollahi et al. "Efficiently Learning Fourier Sparse Set Functions". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. url: https://proceedings.neurips.cc/paper/2019/file/c77331e51c5555f8f935d3344c964bd5-Paper.pdf.

[3] Amir H. Ashouri et al. "A survey on compiler Autotuning using machine learning". In: *ACM Computing Surveys* 51.5 (2019), pp. 1–42. doi: 10.1145/3197978.

[4] Amir Hossein Ashouri et al. "COBAYN: Compiler Autotuning Framework Using Bayesian Networks". In: *ACM Trans. Archit. Code Optim.* 13.2 (June 2016). issn: 1544-3566. doi: 10.1145/2928270. url: https://doi.org/10.1145/2928270.

[5] François Bodin et al. "Iterative Compilation in a Non-linear Optimisation Space". English. In: *Proceedings of the 1998 Workshop on Profile and Feedback Directed Compilation (PFDC'98)*. 1998.

[6] Enrico Brusoni. "Learning Set Functions that are Sparse in Better Non-Orthogonal Fourier Bases". In: ().

[7] Fino and Algazi. "Unified Matrix Treatment of the Fast Walsh-Hadamard Transform". In: *IEEE Transactions on Computers* C-25.11 (1976), pp. 1142–1146. doi: 10.1109/TC.1976.1674569.

[8] M. L. Fisher, G. L. Nemhauser, and L. A. Wolsey. "An analysis of approximations for maximizing submodular set functions—II". In: *Mathematical Programming Studies* (1978), pp. 73–87. doi: 10.1007/bfb0121195.

[9] Carlos Guestrin, Andreas Krause, and Ajit Paul Singh. "Near-optimal sensor placements in Gaussian Processes". In: *Proceedings of the 22nd international conference on Machine learning - ICML '05* (2005). doi: 10.1145/1102351.1102385.

[10] Tierry Hörmann. "Compiler Flag Optimization using Fourier-sparse Poset Functions". 2022.

[11] Markus Puschel, Bastian Seifert, and Chris Wendler. “Discrete signal processing on meet/join lattices”. In: *IEEE Transactions on Signal Processing* 69 (July 2021), pp. 3571–3584. doi: 10.1109/tsp.2021.3081036.

[12] Markus Puschel and Chris Wendler. “Discrete signal processing with set functions”. In: *IEEE Transactions on Signal Processing* 69 (2021), pp. 1039–1053. doi: 10.1109/tsp.2020.3046972.

[13] Lloyd Stowell Shapley. “Notes on the N-person game — II: The value of an N-person game”. In: (1951). doi: 10.7249/rm0670.

[14] Jakob Weissteiner et al. “Fourier analysis-based iterative combinatorial auctions”. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence* (2022). doi: 10.24963/ijcai.2022/78.

[15] Chris Wendler. “Machine learning on non-Euclidean domains: powersets, lattices and posets.” In: ().

[16] Chris Wendler et al. “Learning Set Functions that are Sparse in Non-Orthogonal Fourier Bases”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.12 (May 2021), pp. 10283–10292.